Darwinian Programming and Engineering Design Using Genetic Programming

Forrest H Bennett III
Genetic Programming Inc., Los Altos, California
forrest@evolute.com

John R. Koza
Department of Medicine, Stanford University, Stanford, California
koza@stanford.edu
http://www.smi.stanford.edu/people/koza/

Martin A. Keane
Econometrics Inc., Chicago, Illinois
makeane@ix.netcom.com

David Andre
Computer Science Division, University of California, Berkeley, California
dandre@cs.berkeley.edu

ABSTRACT

One of the central challenges of computer science is to build a system that can automatically create computer programs that are competitive with those produced by humans. This paper presents a candidate set of criteria that identify when a machine-created solution is competitive with a human-produced result. We argue that the field of design is a useful testbed for determining whether an automated technique can produce results that are competitive with human-produced results. We present several results that are competitive with the products of human creativity and inventiveness. This claim is supported by the fact that each of the results infringe on previously issued patents.

1. Introduction

One of the central challenges of computer science is to get a computer to solve a problem without explicitly programming it. In particular, the challenge is to create an automatic system whose input is a high-level statement of a problem's requirements and whose output is a working computer program that solves the given problem. Paraphrasing Arthur Samuel (1959), this challenge concerns

How can computers be made to do what needs to be done, without being told exactly how to do it?

As Samuel (1983) explained,

“The aim [is] ... to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence.”

This paper provides an affirmative answer to the following two questions:

- Can computer programs be automatically created?
- Can automatically created programs be competitive with the products of human creativity and inventiveness?

This paper focuses on a biologically inspired domain-independent technique, called genetic programming, that automatically creates computer programs to solve problems. Starting with a primordial ooze of thousands of randomly created computer programs, genetic programming progressively breeds a population of computer programs over a series of generations using the Darwinian principle of natural selection, recombination (crossover), mutation, gene duplication, gene deletion, and certain mechanisms of developmental biology.

When we talk about a computer program (figure 1), we mean an entity that receives inputs, performs computations, and produces outputs. Computer programs perform basic arithmetic and conditional computations on variables of various types (including integer, floating-point, and Boolean variables), perform iterations and recursions, store intermediate results in memory, organize groups of operations into reusable subroutines, pass information to subroutines in the form of dummy variables (formal parameters), receive information from subroutines in the form of return values, and organize subroutines and a main program into a hierarchy.

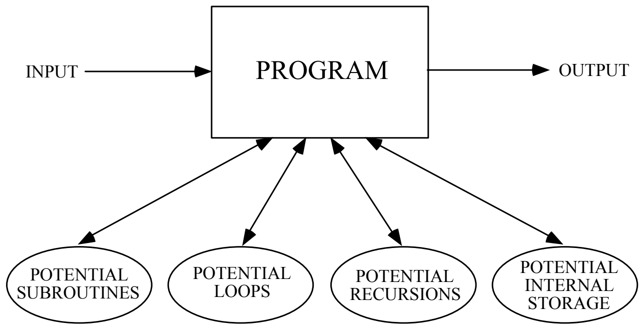

Figure 1 A computer program.

We think that a system for automatically creating computer programs should create entities that possess most or all of the above essential features of computer programs (or reasonable equivalents thereof). A non-definitional list of attributes for a system for automatically creating computer programs would include the following 16 items:

- **Attribute No. 1 (Starts with "What needs to be done"):** It starts from a high-level statement specifying the requirements of the problem.
- **Attribute No. 2 (Tells us "How to do it"):** It produces a result in the form of a sequence of steps that can be executed on a computer.
- **Attribute No. 3 (Produces a computer program):** It produces an entity that can run on a computer.
- **Attribute No. 4 (Automatic determination of program size):** It has the ability to automatically determine the exact number of steps that must be performed and thus does not require the user to prespecify the size of the solution.
- **Attribute No. 5 (Code reuse):** It has the ability to automatically organize useful groups of steps so that they can be reused.
- **Attribute No. 6 (Parameterized reuse):** It has the ability to reuse groups of steps with different instantiations of values (formal parameters or dummy variables).
- **Attribute No. 7 (Internal storage):** It has the ability to use internal storage in the form of single variables, vectors, matrices, arrays, stacks, queues, lists, relational memory, and other data structures.
- **Attribute No. 8 (Iterations, loops, and recursions):** It has the ability to implement iterations, loops, and recursions.
- **Attribute No. 9 (Self-organization of hierarchies):** It has the ability to automatically organize groups of steps into a hierarchy.
- **Attribute No. 10 (Automatic determination of program architecture):** It has the ability to automatically determine whether to employ subroutines, iterations, loops, recursions, and internal storage, and the number of arguments possessed by each subroutine, iteration, loop, recursion.
- **Attribute No. 11 (Wide range of programming constructs):** It has the ability to implement analogs of the programming constructs that human computer programmers find useful, including macros, libraries, typing, pointers, conditional operations, logical functions, integer functions, floating-point functions, complex-valued functions, multiple inputs, multiple outputs, and machine code instructions.
- **Attribute No. 12 (Well-defined):** It operates in a well-defined way. It unmistakably distinguishes between what the user must provide and what the system delivers.
- **Attribute No. 13 (Problem-independent):** It is problem-independent in the sense that the user does not have to modify the system's executable steps for each new problem.
- **Attribute No. 14 (Wide applicability):** It produces a satisfactory solution to a wide variety of problems from many different fields.
- **Attribute No. 15 (Scalability):** It scales well to larger versions of the same problem.
- **Attribute No. 16 (Competitive with human-produced results):** It produces results that are competitive with those produced by human programmers, engineers, mathematicians, and designers.

Conspicuously, the above list of 16 attributes does not preordain that formal logic or an explicit knowledge base must be used to achieve the goal of automatically creating computer programs. Since computer science is founded on logic, many computer scientists unquestioningly assume that formal logic must play a preeminent role in any system for automatically creating computer programs. Similarly, the vast majority of contemporary researchers in artificial intelligence believe that a system for automatically creating computer programs must revolve around an explicit knowledge base. Indeed, over the past four decades, the field of artificial intelligence has been dominated by the strongly-asserted belief that the goal of getting a computer to solve problems automatically can be achieved *only* by means of formal logic and knowledge. This approach typically entails the selection of a knowledge representation, the acquisition of the knowledge, the codification of the knowledge into a knowledge base, the depositing of the knowledge base into a computer, and the manipulation of the knowledge in the computer using formal logic.

Most techniques of artificial intelligence, machine learning, neural networks, adaptive systems, reinforcement learning, or automated logic employ specialized structures in lieu of ordinary computer programs. These surrogate structures include if-then production rules, Horn clauses, decision trees, Bayesian networks, propositional logic, formal grammars, binary decision diagrams, frames, conceptual clusters, concept sets, numerical weight vectors (for neural nets), vectors of numerical coefficients for polynomials or other fixed expressions (for adaptive systems), genetic classifier system rules, fixed tables of values (as in reinforcement learning), or linear chromosome strings (as in the conventional genetic algorithm).

Tellingly, except in unusual situations, the world's several million computer programmers do not use any of these surrogate structures for writing computer programs.

Instead, human programmers write programs that perform arithmetic and conditional computations on variables of various types (including integer, floating-point, and Boolean variables), perform iterations and recursions, store intermediate results in memory, organize groups of operations into reusable subroutines, pass information to and from subroutines, and organize the subroutines and main program into a hierarchy.

All of the above elements of ordinary computer programs have been in use since the beginning of the era of electronic computers in the l940s. Significantly, *none has fallen into disuse by human programmers*. Our view is that if one is really interested in addressing the challenge of getting computers to solve problems without explicitly programming them, then the search should be conducted in the space of computer programs. Moreover, we believe that computer programs are the best representation of computer programs.

2 Criteria for Automatically Produced Results

What do we mean when we say that an automatically created solution to a problem is competitive with the product of human creativity and inventiveness?

We are not referring to the fact that a computer can rapidly print ten thousand payroll checks or that a computer can compute π to a million decimal places. As Fogel, Owens, and Walsh (1966) said,

> Artificial intelligence is realized only if an inanimate machine can solve problems ... not because of the machine's sheer speed and accuracy, but because it can discover for itself new techniques for solving the problem at hand.

We think it is fair to say that an automatically created result is competitive with one produced by human engineers, designers, mathematicians, or programmers if it satisfies any of the following eight criteria (or any other similarly stringent criterion):

(A) The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.

(B) The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed journal.

(C) The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts.

(D) The result is publishable in its own right as a new scientific result (independent of the fact that the result was mechanically created).

(E) The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions.

(F) The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.

(G) The result solves a problem of indisputable difficulty in its field.

(H) The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

3. Genetic Programming

Genetic programming is an extension of the genetic algorithm described in John Holland's pioneering book *Adaptation in Natural and Artificial Systems* (Holland 1975). Genetic programming applies the genetic algorithm to the space of computer programs.

The biological metaphor underlying genetic programming is very different from the underpinnings of all other techniques that have previously been tried in pursuit of the goal of automatically creating computer programs. Many computer scientists and mathematicians are baffled by the suggestion biology might be relevant to solving important problems in their fields. However, we do not view biology as an unlikely well from which to draw a solution to the challenge of getting a computer to solve a problem without explicitly programming it. Quite the contrary – we view biology as a most likely source. Indeed, genetic programming is based on the only method that has ever produced intelligence – the time-tested method of evolution and natural selection.

Of course, we did not originate the idea that machine intelligence may be realized using a biological approach. Turing made the connection between searches and the challenge of getting a computer to solve a problem without explicitly programming it in his 1948 essay "Intelligent Machines" (Ince 1992).

> Further research into intelligence of machinery will probably be very greatly concerned with “searches” ...

Turing then identified three broad approaches by which search might be used to automatically create an intelligent computer program.

One approach that Turing identified is a search through the space of integers representing candidate computer programs. This approach, of course, uses many of the techniques that Turing used in his own work on the foundations of computation.

Another approach is the "cultural search" which relies on knowledge and expertise acquired over a period of years from others. This approach is akin to present-day knowledge-based systems.

The third approach that Turing specifically identified is “genetical or evolutionary search.” Turing said,

> There is the genetical or evolutionary search by which a combination of genes is looked for, the criterion being the survival value. The remarkable success of this search confirms to some extent the idea that intellectual activity consists mainly of various kinds of search.

Turing did not specify how to conduct the "genetical or evolutionary search" for a computer program. However, his 1950 paper "Computing Machinery and Intelligence" (Ince 1992) suggested how natural selection and evolution might be incorporated into the search for intelligent machines.

> We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications
>
> Structure of the child machine = Hereditary material
>
> Changes of the child machine = Mutations
>
> Natural selection = Judgment of the experimenter

3.1 Implementation of Turing's Third Way

Genetic programming implements Turing’s third way to achieve machine intelligence. Specifically, genetic programming starts with an initial population (generation 0) of randomly generated computer programs composed of the given primitive functions and terminals. The programs in the population are, in general, of different sizes and shapes. The creation of the initial random population is a blind random search of the space of computer programs composed of the problem's available functions and terminals.

On each generation of a run of genetic programming, each individual in the population of programs is evaluated as to its fitness in solving the problem at hand. The programs in generation 0 of a run almost always have exceedingly poor fitness for non-trivial problems of interest. Nonetheless, some individuals in a population will turn out to be somewhat more fit than others. These differences in performance are then exploited so as to direct the remainder of the search into promising areas of the search space. The Darwinian principle of reproduction and survival of the fittest is used to probabilistically select, on the basis of fitness, individuals from the population to participate in various operations. A small percentage (e.g., 9%) of the selected individuals are reproduced (copied) from one generation to the next. A very small percentage (e.g. 1%) of the selected individuals are mutated in a random way. About 90% of the selected individuals participate in the genetic operation of crossover (sexual recombination) to create offspring programs by recombining genetic material from two parents. All operations are performed so as to create offspring that are syntactically valid and executable. After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the old population (i.e., the old generation). Then, each individual in the new population of programs is measured for fitness, and this iterative process is repeated over many generations.

Probabilistic steps are pervasive in genetic programming. Probability is involved in the creation the individuals in the initial population, the selection of individuals to participate in the operations of reproduction, crossover, and mutation, and the selection of crossover and mutation points within parental programs.

The dynamic variability of the size and shape of the computer programs that are created during the run is an important feature of genetic programming. It is often difficult and unnatural to try to specify or restrict the size and shape of the eventual solution in advance.

Additional information on current research in genetic programming can be found in *Genetic Programming III: Darwinian Invention and Problem Solving* (Koza, Bennett, Andre, and Keane 1999a) and the accompanying videotape (Koza, Bennett, Andre, Keane, and Brave 1999b) and in Koza 1992; Koza and Rice 1992,; Koza 1994a; Koza 1994b; Banzhaf, Nordin, Keller, and Francone 1998; Langdon 1998; Kinnear 1994; Angeline and Kinnear 1996; Spector, Langdon, O'Reilly, and Angeline 1999; Koza, Goldberg, Fogel, and Riolo 1996; Koza, Deb, Dorigo, Fogel, Garzon, Iba, and Riolo 1997; Koza, Banzhaf, Chellapilla, Deb, Dorigo, Fogel, Garzon, Goldberg, Iba, and Riolo 1998; Banzhaf, Daida, Eiben, Garzon, Honavar, Jakiela, and Smith 1999; Banzhaf, Poli, Schoenauer, and Fogarty 1998; and Poli, Nordin, Langdon, and Fogarty 1999.

4. Design as a Testbed for Automatically Produced Results

Design is a major activity of practicing engineers. The design process entails creation of a complex structure to satisfy user-defined requirements. Since the design process typically entails tradeoffs between competing considerations, the end product of the process is usually a satisfactory and compliant design as opposed to a perfect design. Design is usually viewed as requiring creativity and human intelligence. Consequently, the field of design is a source of challenging problems for automated techniques of machine intelligence. In particular, design problems are useful for determining whether an automated technique can produce results that are competitive with human-produced results.

The design (synthesis) of analog electrical circuits is especially challenging. The design process for analog circuits begins with a high-level description of the circuit's desired behavior and characteristics and entails creation of both the topology and the sizing of a satisfactory circuit. The topology comprises the gross number of components in the circuit, the type of each component (e.g., a capacitor), and a list of all connections between the components. The sizing involves specifying the values (typically numerical) of each of the circuit's components.

Although considerable progress has been made in automating the synthesis of certain categories of purely digital circuits, the synthesis of analog circuits and mixed analog-digital circuits has not proved to be as amenable to automation. There is no previously known general technique for automatically creating an analog circuit from a high-level statement of the design goals of the circuit. As O. Aaserud and I. Ring Nielsen (1995) observe,

> Analog designers are few and far between. In contrast to digital design, most of the analog circuits are still handcrafted by the experts or so-called 'zahs' of analog design. The design process is characterized by a combination of experience and intuition and requires a thorough knowledge of the process characteristics and the detailed specifications of the actual product.
>
> Analog circuit design is known to be a knowledge-intensive, multiphase, iterative task, which usually stretches over a significant period of time and is performed by designers with a large portfolio of skills. It is therefore considered by many to be a form of art rather than a science.

This paper focuses on three particular problems of analog circuit synthesis, namely the design of a lowpass filter circuit, the design of a high-gain, low-distortion, low-bias amplifier, and the design of a cube root computational circuit.

A simple analog *filter* is a one-input, one-output circuit that receives a signal as its input and passes the frequency components of the incoming signal that lie in a specified range (called the *passband*) while suppressing the frequency components that lie in all other frequency ranges (the *stopband*). Specifically, the goal is to design a lowpass filter composed of capacitors and inductors that passes all frequencies below 1,000 Hertz (Hz) and suppresses all frequencies above 2,000 Hz.

An amplifier is a one-input, one-output circuit whose output is a constant multiple of its input. We are seeking a high-gain, low-distortion, low-bias amplifier composed of transistors, diodes, capacitors, resistors, and connections to power sources.

An analog computational circuit is a one-input, one-output circuit whose output is a specified mathematical function. The design of computational circuits is exceedingly difficult even for seemingly mundane mathematical functions. Success often relies on the clever exploitation of some aspect of the underlying device physics of the components that is unique to the particular desired mathematical function. Because of this, the implementation of each different mathematical function typically requires an entirely different clever insight and an entirely different circuit. We are seeking a computational circuit composed of transistors, diodes, capacitors, resistors, and connections to power sources.

It should be noted that the approach described in this paper has also been successfully applied to numerous other problems of analog circuit synthesis, including the design of a temperature-sensing circuit, a voltage reference circuit, a time-optimal robot controller circuit, a difficult-to-design asymmetric bandpass filter, crossover filters, a double passband filter, bandstop filters, highpass filters, frequency discriminator circuits, a frequency-measuring circuit, other amplifiers, and other computational circuits.

5. Applying Genetic Programming to Circuit Synthesis

Genetic programming can be applied to the problem of synthesizing circuits if a mapping is established between the program trees (rooted, point-labeled trees with ordered branches) used in genetic programming and the labeled cyclic graphs germane to electrical circuits. The principles of developmental biology provide the motivation for mapping trees into circuits by means of a developmental process that begins with a simple embryo. For circuits, the embryo typically includes fixed wires that connect the inputs and outputs of the particular circuit being designed and certain fixed components (such as source and load resistors). Until these wires are modified, the circuit does not produce interesting output. An electrical circuit is developed by progressively applying the functions in a circuit-constructing program tree to the modifiable wires of the embryo (and, during the developmental process, to new components and modifiable wires).

An electrical circuit is created by executing the functions in a circuit-constructing program tree. The functions are progressively applied in a developmental process to the embryo and its successors until all of the functions in the program tree are executed. That is, the functions in the circuit-constructing program tree progressively side-effect the embryo and its successors until a fully developed circuit eventually emerges. The functions are applied in a breadth-first order.

The functions in the circuit-constructing program trees are divided into five categories: (1) topology-modifying functions that alter the circuit topology, (2) component-creating functions that insert components into the circuit, (3) development-controlling functions that control the development process by which the embryo and its successors is changed into a fully developed circuit, (4) arithmetic-performing functions that appear in subtrees as argument(s) to the component-creating functions and specify the numerical value of the component, and (5) automatically defined functions that appear in the automatically defined functions and potentially enable certain substructures of the circuit to be reused (with parameterization).

Before applying genetic programming to a problem of circuit design, seven major preparatory steps are required: (1) identify the embryonic circuit, (2) determine the architecture of the circuit-constructing program trees, (3) identify the primitive functions of the program trees, (4) identify the terminals of the program trees, (5) create the fitness measure, (6) choose control parameters for the run, and (7) determine the termination criterion and method of result designation.

A detailed discussion concerning how to apply these seven preparatory steps to particular problems is found in Koza, Bennett, Andre, and Keane 1999a (chapter 25).

6. Results on Illustrative Problems

6. 1 Campbell 1917 Ladder Filter Patent

The best circuit (figure 2) of generation 49 of one run of genetic programming on the problem of synthesizing a lowpass filter is a 100% compliant circuit.

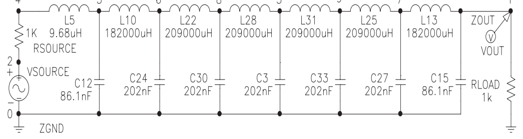

Figure 2 Evolved Campbell filter.

The evolved circuit is what is now called a cascade (ladder) of identical π sections and is shown and analyzed in Koza, Bennett, Andre, and Keane 1999a (chapter 25). The evolved circuit has the recognizable topology of the circuit for which George Campbell of American Telephone and Telegraph received U. S. patent 1,227,113 in 1917. In addition to possessing the topology of the Campbell filter, the numerical value of all the components in the evolved circuit closely approximate the numerical values specified in Campbell's 1917 patent. But for the fact that this 1917 patent has expired, the evolved circuit would infringe on the Campbell patent.

The fact that genetic programming rediscovered both the topology and sizing of an electrical circuit that was unobvious "to a person having ordinary skill in the art" establishes that this evolved result satisfies Arthur Samuel's criterion for artificial intelligence and machine learning (quoted in section 1).

Since filing for a patent entails the expenditure of a considerable amount of time and money, patents are generally sought, in the first place, only if an individual or business believes the inventions are likely to be useful in the real world and economically rewarding. Patents are only issued if an arms-length examiner is convinced that the proposed invention is novel, useful, and satisfies the statutory test for unobviousness.

6.2 Zobel 1925 "*M*-Derived Half Section" Patent

In another run of this same problem of synthesizing a lowpass filter, a 100%-compliant circuit (figure 3) was evolved in generation 34.

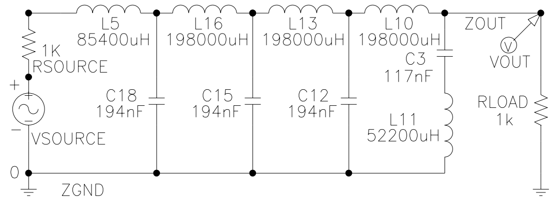

Figure 3 Evolved Zobel filter.

This evolved circuit (presented in Koza, Bennett, Andre, and Keane 1999a, chapter 25) is equivalent to a cascade of three symmetric T-sections and an *M*-derived half section. Otto Zobel of American Telephone and Telegraph Company invented the idea of adding an "*M*-derived half section" to one or more "constant K" sections.

6.3 Cauer 1934 – 1936 Elliptic Patents

In yet another run of this same problem of synthesizing a lowpass filter, a 100% compliant circuit (figure 4) emerged in generation 31 (Koza, Bennett, Andre, and Keane 1999a, chapter 27).

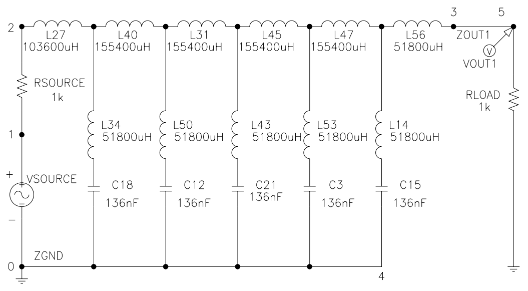

Figure 4 Evolved Cauer (elliptic) filter topology.

This circuit has the recognizable elliptic topology that was invented and patented by Wilhelm Cauer in 1934, 1935, and 1936. The Cauer filter was a significant advance (both theoretically and commercially) over the earlier filter designs of Campbell, Zobel, Johnson, Butterworth, and Chebychev. For example, for one commercially important set of specifications for telephones, a fifth-order elliptic filter matches the behavior of a 17th-order Butterworth filter or an eighth-order Chebychev filter. The fifth-order elliptic filter has one less component than the eighth-order Chebychev filter. As Van Valkenburg (1982) relates in connection with the history of the elliptic filter:

> Cauer first used his new theory in solving a filter problem for the German telephone industry. His new design achieved specifications with one less inductor than had ever been done before. The world first learned of the Cauer method not through scholarly publication but through a patent disclosure, which eventually reached the Bell Laboratories. Legend has it that the entire Mathematics Department of Bell Laboratories spent the next two weeks at the New York Public library studying elliptic functions. Cauer had studied mathematics under Hilbert at Goettingen, and so elliptic functions and their applications were familiar to him.

Genetic programming did not, of course, study mathematics under Hilbert or anybody else. Instead, the elliptic topology emerged from a run of genetic programming as a natural consequence of the problem's fitness measure and natural selection – not because the run was primed with domain knowledge about elliptic functions or filters or electrical circuitry. Genetic programming opportunistically *reinvented* the elliptic topology because necessity (fitness) is the mother of invention.

6.4 Darlington 1952 Emitter-Follower Patent

Sidney Darlington of the Bell Telephone Laboratories obtained some 40 patents on numerous fundamental electronic circuits. In particular, he obtained U. S. patent 2,663,806 for what is now called the Darlington emitter-follower section. We have evolved Darlington emitter-follower sections on 12 occasions in the process of solving problems of analog circuit synthesis.

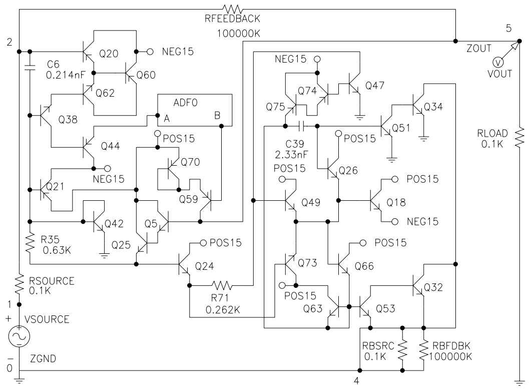

Figure 5 Evolved 96 dB amplifier.

For example, figure 5 shows the best circuit from generation 86 of a run of the problem of evolving a high-gain, low-distortion, low-bias amplifier. The circuit has 25 transistors, no diodes, two capacitors, and two resistors and contains a Darlington emitter-follower section (involving transistors Q25 and Q5).

As another example, figure 6 shows the best-of-run circuit from generation 57 of the problem of synthesizing a cube root computational circuit. The circuit has 38 transistors, seven diodes, and 18 resistors.

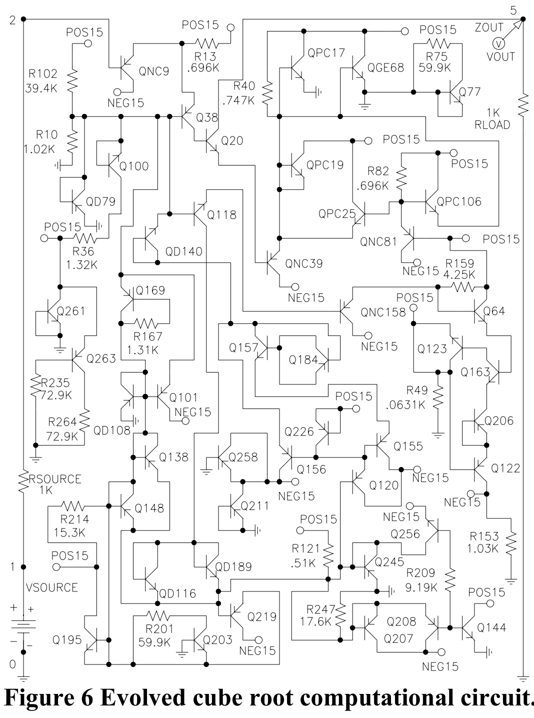

Figure 6 Evolved cube root computational circuit.

7 Additional Results

Table 1 shows 14 instances of results where genetic programming has produced results that are competitive with the products of human creativity and inventiveness (Koza, Bennett, Andre, and Keane 1999a). Each claim is accompanied by the particular criterion (from section 2) that establishes the basis for the claim. The instances in the table include classification problems from the field of computational molecular biology, a long-standing problem involving cellular automata, a problem of synthesizing the design of a minimal sorting network, and several problems of synthesizing the design of analog electrical circuits. As can be seen, 10 of the 14 instances in the table involve previously patented inventions. The evolved results for claim instances 1, 2, 13, and 14 in the table used architecture altering operations (Koza, Bennett, Andre, and Keane 1999a) to automatically determine that subroutines were useful in solving the problem.

Table 1 Fourteen instances where genetic programming has produced results that are competitive with human-produced results.

	Claimed instance	Basis for claim
1	Creation of four different algorithms for the transmembrane segment identification problem for proteins	B, E
2	Creation of a sorting network for seven items using only 16 steps	A, D
3	Rediscovery of the Campbell ladder topology for lowpass and highpass filters	A, F
4	Rediscovery of "*M*-derived half section" and "constant *K*" filter sections	A, F
5	Rediscovery of the Cauer (elliptic) topology for filters	A, F
6	Automatic decomposition of the problem of synthesizing a crossover filter	A, F
7	Rediscovery of a recognizable voltage gain stage and a Darlington emitter-follower section of an amplifier and other circuits	A, F
8	Synthesis of 60 and 96 decibel amplifiers	A, F
9	Synthesis of analog computational circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions	A, D, G
10	Synthesis of a real-time analog circuit for time-optimal control of a robot	G
11	Synthesis of an electronic thermometer	A, G
12	Synthesis of a voltage reference circuit	A, G
13	Creation of a cellular automata rule for the majority classification problem that is better than the Gacs-Kurdyumov-Levin (GKL) rule and all other known rules written by humans	D, E
14	Creation of motifs that detect the D–E–A-D box family of proteins and the manganese superoxide dismutase family	C

References

Aaserud, O. and Nielsen, I. Ring. 1995. Trends in current analog design: A panel debate. *Analog Integrated Circuits and Signal Processing*. 7(1) 5-9.

Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Banzhaf, Wolfgang, Daida, Jason, Eiben, A. E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E. (editors). 1999. *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida USA*. San Francisco, CA: Morgan Kaufmann.

Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. 1998. *Genetic Programming – An Introduction*. San Francisco, CA: Morgan Kaufmann and Heidelberg: dpunkt.

Banzhaf, Wolfgang, Poli, Riccardo, Schoenauer, Marc, and Fogarty, Terence C. 1998. *Genetic Programming: First European Workshop. EuroGP'98. Paris, France, April 1998 Proceedings. Paris, France. April l998*. Lecture Notes in Computer Science. Volume 1391. Berlin, Germany: Springer-Verlag.

Campbell, George A. 1917. *Electric Wave Filter*. Filed July 15, 1915. U. S. Patent 1,227,113. Issued May 22, 1917.

Cauer, Wilhelm. 1934. *Artificial Network*. U. S. Patent 1,958,742. Filed June 8, 1928 in Germany. Filed December 1, 1930 in United States. Issued May 15, 1934.

Cauer, Wilhelm. 1935. *Electric Wave Filter*. U. S. Patent 1,989,545. Filed June 8, 1928. Filed December 6, 1930 in United States. Issued January 29, 1935.

Cauer, Wilhelm. 1936. *Unsymmetrical Electric Wave Filter*. Filed November 10, 1932 in Germany. Filed November 23, 1933 in United States. Issued July 21, 1936.

Fogel, Lawrence J., Owens, Alvin J., and Walsh, Michael. J. 1966. *Artificial Intelligence through Simulated Evolution*. New York: John Wiley.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

Ince, D. C. (editor). 1992. *Mechanical Intelligence: Collected Works of A. M. Turing*. Amsterdam: North Holland.

Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

Koza, John R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: MIT Press.

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: MIT Press.

Koza, John R., Banzhaf, Wolfgang, Chellapilla, Kumar, Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max H., Goldberg, David E., Iba, Hitoshi, and Riolo, Rick. (editors). 1998. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1999a. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., Bennett III, Forrest H, Andre, David, Keane, Martin A., and Brave, Scott. 1999b. *Genetic Programming III Videotape*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference* San Francisco, CA: Morgan Kaufmann.

Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: The MIT Press.

Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: MIT Press.

Langdon, William B. 1998. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Amsterdam: Kluwer.

Poli, Riccardo, Nordin, Peter, Langdon, William B., and Fogarty, Terence C. 1999. *Genetic Programming: Second European Workshop. EuroGP'99. Goteborg, Sweden, May 1999, Proceedings*. Lecture Notes in Computer Science. Volume 1598. Berlin, Germany: Springer-Verlag.

Samuel, Arthur L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development.* 3(3): 210–229.

Samuel, Arthur L. 1983. AI: Where it has been and where it is going. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*. Los Altos, CA: Morgan Kaufmann. Pages 1152 – 1157.

Spector, Lee, Langdon, William B., O'Reilly, Una-May, and Angeline, Peter (editors). 1999. *Advances in Genetic Programming 3*. Cambridge, MA: The MIT Press.

Van Valkenburg, M. E. 1982. *Analog Filter Design*. Fort Worth, TX: Harcourt Brace Jovanovich.

Zobel, Otto Julius. 1925. *Wave Filter*. Filed January 15, 1921. U. S. Patent 1,538,964. Issued May 26, 1925.

Version 1 – March 15, 1999 for SCASE workshop in Leeds, Ireland in April, 1999.

Darwinian Programming and Engineering Design Using Genetic Programming

Forrest H Bennett III
Chief Scientist
Genetic Programming Inc.
Box 1669
Los Altos, California 94023
forrest@evolute.com
http://www.genetic-programming.com

John R. Koza
Section on Medical Informatics
Department of Medicine
Medical School Office Building
Stanford University
Stanford, California 94305
koza@stanford.edu
http://www.smi.stanford.edu/people/koza/

Martin A. Keane
Chief Scientist
Econometrics Inc.
111 E. Wacker Dr.
Chicago, Illinois 60601
makeane@ix.netcom.com

David Andre
Division of Computer Science
University of California
Berkeley, California 94720
dandre@cs.berkeley.edu